

Byte-Range-Locks using MPI-One-Sided Communication: a report

Salman Pervez

March 15, 2006

Abstract

This paper presents a case study on detecting and fixing a subtle error in the byte-range locking algorithm introduced in [1]. We use model-checking to identify the error and introduce various versions each improving upon the severity of the error with the final version eliminating it altogether. Preliminary analysis shows that the error-free version of the algorithm performs comparably to the original version.

1 Introduction

MPI, or message passing interface, is a paradigm used mainly for writing programs that run on parallel machines with distributed memory. The MPI-2.0 standard introduces One-Sided communication which gives the programmer the ability to write shared-memory programs in a distributed memory environment.

Writing parallel programs in any setting is hard. Debugging them is even harder. Classical errors such as deadlocks and livelocks are very common and difficult to locate. Certain program orderings hide these errors during preliminary testing but they are exposed when run under a different ordering. Model checking can be used to effectively catch and remove such errors by exploring all possible program orderings for a given algorithm.

In this paper we use the Spin model checker to verify a promela model for the byte-range locking algorithm presented in [1]. We show how the model exposed a subtle error pertaining to errant Send's and differet versions of the algorithm that reduces the numner of errant Send's.

2 The Original Algorithm

The first step is to define a memory model suitable for the protocol. The authors propose a model in which one process makes its memory space available for communication purposes, we will call this memory region M. All inter-process communication now takes place through reads and writes to M. Space is reserved for each process to make available three pieces of information, a flag value(0/1)

and Start and End values indicating the range for which a lock is desired. This information determines the state of the process at any given time in the protocol. In other words, if process A needs to find process B's state, all it can do is read these three values from the shared memory location M.

3 One-Sided constructs used

The one-sided communication primitives used in this protocol are presented below. The semantics for these calls specify a set of behaviours that is a subset of those specified in the MPI Standard. In other words, any errors detected when following these semantics will also be observed when using the original semantics.

3.1 `MPI_Win_lock(MPI_LOCK_EXCLUSIVE, ..., M)`

This call is made by a process indicating that it needs to access the shared memory region for reads or writes. This is a blocking call which does not return until permission is granted for this process to access M. Note the use of the keyword `MPI_LOCK_EXCLUSIVE`. MPI semantics dictate that the process will now have exclusive access to M. No other processes can read or write from/to M until this process calls `MPI_Win_unlock`.

Note that these semantics differ from those specified in the MPI-2.0 standard. The standard states that the lock may be acquired at any statement up until the `MPI_Win_unlock` statement. However, an error detected using these semantics must also be a valid error in MPI-2.0 according to the standard.

3.2 `MPI_Win_unlock(..., M)`

This call is made by a process to indicate that it has finished reading from/writing to M. Other requests for access to M may now be processed.

3.3 `MPI_Put(value, ..., dest, ..., M)`

This call is made to write 'value' to M. The parameter `dest` indicates where 'value' is to be written. Note that a process can make this call to write to another processes' memory space without any actions taken by the target process. This is called passive communication and is allowed only under the `MPI_Win_lock/MPI_Win_unlock` synchronization scheme. In this protocol, this call is specifically used to write the flag, start and end values of a process into M changing M's global value.

3.4 `MPI_Get(buf, src, ..., M)`

This call is made to read the contents of M into `buf`. The calling process can specify which part of M's memory to read using suitable derived datatypes. In

this algorithm, this call is made to read the global state of all processes except for the state of the calling process.

4 The Protocol

```
1 Lock_acquire ( int start , int end)
2 {
3   val [0] = 1; /* flag */ val [1] = start ; val [2] = end ;
4   while (1) {
5     /* add self to locklist */
6     MPI_Win_lock ( MPI_LOCK_EXCLUSIVE , homerank , 0, lockwin );
7     MPI_Put (& val , 3, MPI_INT , homerank , 3*( myrank ),
8             3, MPI_INT , lockwin );
9     MPI_Get ( locklistcopy , 3*( nprocs -1) , MPI_INT , homerank ,
10            0, 1, locktype1 , lockwin );
11    MPI_Win_unlock ( homerank , lockwin );
12    /* check to see if lock is already held */
13    conflict = 0;
14    for (i=0; i < ( nprocs - 1); i ++ ) {
15      if (( flag == 1) && ( byte ranges conflict with lock request )) {
16        conflict = 1; break ;
17      }
18    }
19    if ( conflict == 1) {
20      /* reset flag to 0, wait for notification , and then retry the lock */
21      MPI_Win_lock ( MPI_LOCK_EXCLUSIVE , homerank , 0, lockwin );
22      val [0] = 0;
23      MPI_Put (val , 1, MPI_INT , homerank , 3*( myrank ), 1, MPI_INT ,
24              lockwin );
25      MPI_Win_unlock ( homerank , lockwin );
26      /* wait for notification from some other process */
27      MPI_Recv (NULL , 0, MPI_BYTE , MPI_ANY_SOURCE , WAKEUP , comm ,
28               MPI_STATUS_IGNORE );
29      /* retry the lock */
30    }
31    else {
32      /* lock is acquired */
33      break ;
34    }
35  }
36 }
```

```

34 Lock_release ( int start , int end)
35 {
36     val [0] = 0; val [1] = -1; val; [2] = -1;
37     /* set start and end offsets to -1, flag to 0, and get everyone
        else 's status */
38     MPI_Win_lock ( MPI_LOCK_EXCLUSIVE , homerank , 0, lockwin );
39     MPI_Put ( val , 3, MPI_INT , homerank , 3*( myrank ), 3, MPI_INT , lockwin )
40     MPI_Get ( locklistcopy , 3*( nprocs -1) , MPI_INT , homerank ,
        0, 1, locktype2 , lockwin );
41     MPI_Win_unlock ( homerank , lockwin );
42     /* check if anyone is waiting for a conflicting lock . If so , send them a
43     0- byte message , in response to which they will retry the lock . For
44     fairness , we start with the rank after ours and look in order . */
45     i = myrank ; /* ranks are off by 1 because of the derived datatype */
46     while ( i < ( nprocs - 1)) {
47         /* the flag doesn t matter here . check only the byte ranges */
48         if ( byte ranges conflict ) MPI_Send ( NULL , 0, MPI_BYTE , i+1,
            WAKEUP , comm );
49         i++;
50     }
51     i = 0;
52     while ( i < myrank ) {
53         if ( byte ranges conflict ) MPI_Send ( NULL , 0, MPI_BYTE , i ,
            WAKEUP , comm );
54         i++;
55     }
56 }

```

The protocol consists of two parts. There is the lock acquire part and the lock release part. During the acquire phase, a process first registers itself as a potential lock-owner showing its intent to eventually acquire the lock. It does this in lines 6-9 where it simultaneously writes its flag, start and end values to M and reads every other process' current state. It now checks whether or not its desired lock range conflicts with another process'. If so, it backs off, sets its flag bit to 0 and blocks on the MPIRecv call on line 24. If there is no conflict, the lock has been acquired and the process may safely enter the critical section.

The lock release part is also very simple. The process with the lock must first free the lock by writing the appropriate values (0, -1, -1) to M. It also simultaneously reads the global state of all other processes. It must now check to see if any of the processes is/will be blocked on the lock it just had. If so, it must issue an MPI_Send request to wake up that process. Once it has woken up all the blocked processes it can return from the lock_release subroutine. The blocked processes that it just sent messages to must now go back to line 4 and try to re-acquire the lock.

4.1 The Error

There is a subtle error in this routine. The error arises from the fact that at no point does any process know the exact global state of any other process. The information it reads at any point in time cannot be relied upon at any other point in time. The only guarantee we have is that the information we acquired was true some time before now. The protocol uses this fact effectively to acquire the lock since the acquire step is based on "past information" and not necessarily "present information". For instance, if P1 and P2 both attempt to acquire the lock and P1 sees P2's global state as such, P1 can assume that P2 tried to acquire the lock before me (past information) and will back off given this information. It does not make any assumptions about the current state of P2. For all it knows, P2 may already have acquired the lock or may be blocked on the MPI_Recv call. Although this works out nicely in the acquire routine, we are not as lucky in the release routine.

In the release routine, a process reads the current global state of the system in lines 38-41. It then goes on to examine this global state in lines 48 and 53. It then makes a decision based on this information. The decision it makes is whether or not a process is/will be blocked because I had the lock that it wanted. If so, I must issue an MPI_Send request to make sure that process is not blocked forever. Note that in this case, the process releasing the lock assumes that the process it is trying to wake up is still blocked. However, as discussed earlier, there is no guarantee that this process will still be blocked. It was blocked when the information was first acquired, but the state may have changed by now.

The error is simply that processes send many more MPI_Send requests than there are available MPI_Recv requests. An error trace of a specific error situation is presented below.

Suppose there are three processes trying to acquire a lock. Call these P1, P2 and P3. The error trace in Figure 1 shows where a particular process is in its program execution compared to the other processes. P1 tries to acquire a lock on the range (3, 5) and succeeds since no other process has a lock on this range. Similarly, P2 successfully acquires a lock on the range (6, 8). P3 then tries to acquire a lock on the range (5, 6). It fails since its desired range conflicts with both P1 and P2. So P3 blocks on the MPI_Recv call. P2 releases the lock and Sends a message to P3. P3 will now re-acquire the lock. The error occurs when P1 now tries to Send a message to P3. It does so since P3 has not yet had a chance to change its global state and P1 erroneously assumes that P3 is still blocked.

The error scenario described above results in two MPI_Send calls being issued to P3. P3 only receives one of these while the other one stays in the system. Two things can go wrong in this case. Depending on the MPI implementation being used, P1 may be blocked on the MPI_Send call and may not be unblocked ever. Assuming that the MPI implementation does not block on 0-byte messages, we will now have an extra send request in the system which will contribute to memory leaks. This may become a factor in a memory intensive program where there is high contention for locks. For instance, in an n process system, n - 1

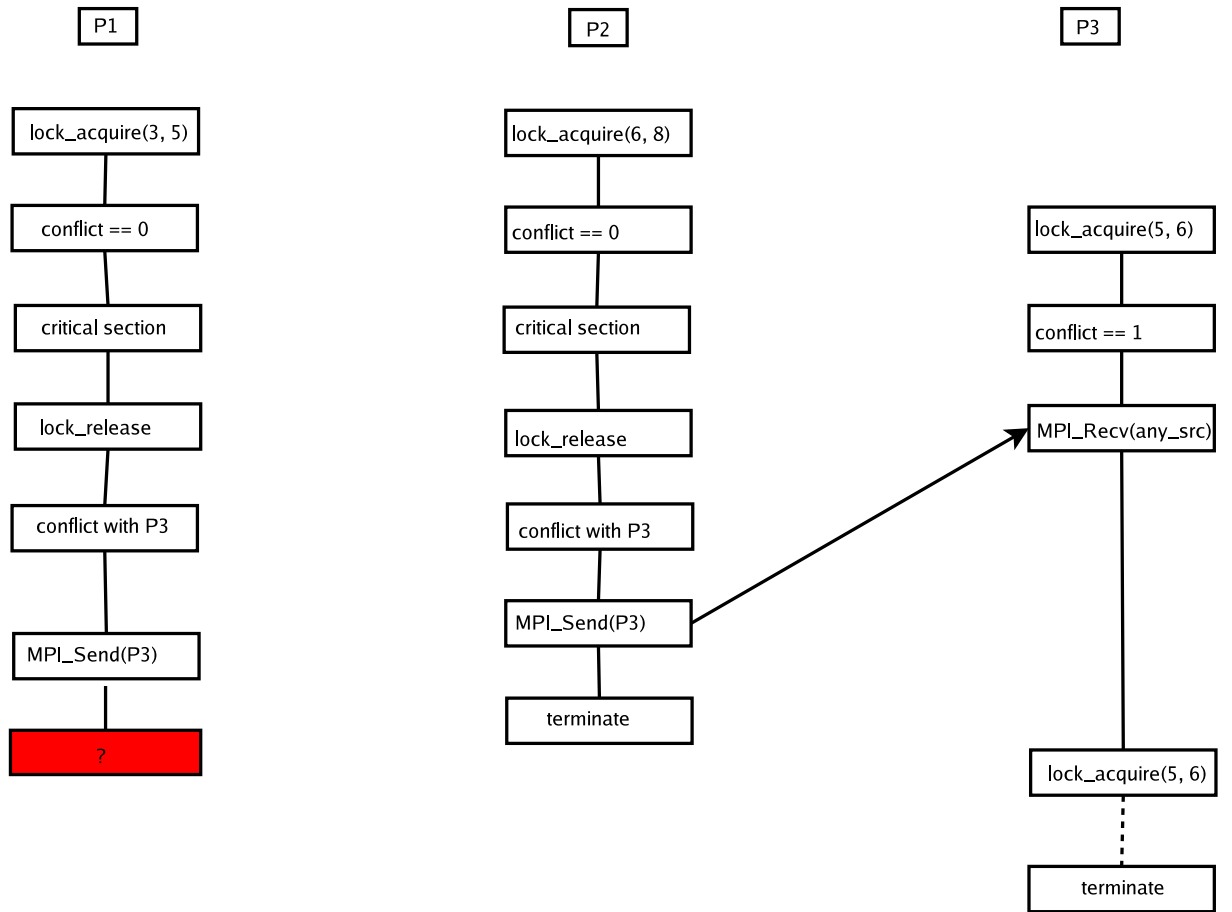


Figure 1: Original Error Scenario

processes might acquire the lock on non-intersecting ranges. The n th process may now request a lock on the entire byte-range. This may result in $n-2$ errant Send messages to accumulate in the system. It is easy to see that this number is theoretically unbounded.

This bug was caught using a promela model verified using the Spin model checker. The model accurately reflects the semantics of MPI one-sided communication, given constraints described above, and took less than a second to find the bug.

5 Possible Solution

```

1 Lock_release ( int start , int end)
2 {
3   val [0] = 0; val [1] = -1; val [2] = -1;
4
5   int myBids[nprocs];
6   int globalBids[nprocs][nprocs];
7
8   /* set start and end offsets to -1, flag to 0, and get
       everyone else 's status */
9   MPI_Win_lock ( MPI_LOCK_EXCLUSIVE , homerank , 0, lockwin );
10  MPI_Put (val , 3, MPI_INT , homerank , 3*( myrank ), 3, MPI_INT ,
        lockwin );
11  MPI_Get ( locklistcopy , 3*( nprocs -1) , MPI_INT , homerank ,
        0, 1, locktype2 , lockwin );
12  MPI_Win_unlock ( homerank , lockwin );
13
14  for(int i = 0; i < nprocs; i++) {
15    /* figure out which blocked processes I want to bid on */
16    if ( byte ranges conflict ) myBids[i] = 1;
17  }
18
19  MPI_Win_lock ( MPI_LOCK_EXCLUSIVE, homerank, 0, lokcwin);
20  /* Put your bids into B using a suitable derived datatype.
       Note that these will be placed in a coloumn, not in a row.*/
21  MPI_Put (myBids, nprocs, MPI_INT, homerank, B[i]'s address,
        nprocs, MPI_INT, lockwin);
22  /* Read the current contents of B into private memory */
23  MPI_Get(globalBids, ..., B's address, ..., lockwin);
24  MPI_Win_unlock ( homerank , lockwin );
25
26  /* for each process , check if I am the only bidder.
       If so, I must wake up that process.
```

```

    If not, I must back off */
27 for(int i = 0; i < nprocs; i++) {
28
29     if(B[i][j] == 0 for all j != myrank) {
30         MPI_Send (NULL , 0, MPI_BYTE , i, WAKEUP , comm );
31
32         MPI_Win_lock ( MPI_LOCK_EXCLUSIVE , homerank , 0, lockwin );
33         /* Place 0's in the bidding array B[i] */
34         MPI_Put (array of 0's , nprocs, MPI_INT , homerank , B[i]'s address ,
                 nprocs, MPI_INT , lockwin );
35         MPI_Win_unlock ( homerank , lockwin );
36     }
37 }
38}

```

A cursory examination of the error trace reveals that the problem is related to the fact that we have more than one processes trying to wake up the same blocked process. As described, the reason for this is that the information used to detect (potentially) blocked processes has gone stale and cannot be trusted. One way to fix this would be to make sure that each blocked process is woken up by only 1 other process. The fix focuses on the fact that there can be multiple processes in the critical section at the same time. When they leave the critical section, they might all try to wake up the same blocked process which needs to be avoided. The proposed solution involves a bidding process in which each process bids on a blocked process that it wants to wake up. The quickest bidder gets the right to wake up the blocked process. The rest back off.

To implement the fix we must add a bidding array B to M. This is an $N \times N$ array where N is the number of processes in the program. Each row of N entries corresponds to a unique process with the same pid. Suppose we have N processes $P_0 \dots P_{n-1}$. Each P_i has a corresponding bidding array $B[i]$. Now suppose that processes 0 through $n-2$ are each in the critical section while process $n-1$ is blocked on the entire range. Under the original protocol, once P_0 through P_{n-2} leave their critical section they may each issue an `MPI_Send` for P_{n-1} . Under the new protocol, they will each have to bid on the right to wake up P_{n-1} . The bidding process is explained in the pseudocode (lines 10-24). It consists of simply writing a 1 in $B[i][j]$ where i is the blocked process being bid on and j is the process that does the bidding. The winning process, i.e. the only process p that sees $B[i][j] == 0$ for all $j \neq p$, is the one that gets to wake up i . This will eliminate the error where multiple processes mistakenly send messages to the same blocked process.

Note that none of the processes need know what the "current" state of the others processes is. All they know is that none of the others will issue an `MPI_Send` message to the process they are waking up. This fix was also run through the protocol's promela model and yet another error was found. Once again, this error arose due to the fact that one of the processes did not know what the current state of the other processes was and relied on stale information

to make a conclusion. Note that in line 34 of the fixed code, the highest bidding process must place 0's in the global bidding array. The decision to place 0's at this point in the program is arbitrary. The confusion arises when a potential bidder is too slow in getting to the global bidding array and read it only after the winner has replaced the original bidding array with 0's. This slow process now thinks that it was the first to bid on the blocked process and also issues an MPI_Send. It would seem that we are back to square 1.

A deeper analysis of this solution reveals that although extra Send's are still possible, their number is far lesser. As discussed earlier, deadlock can be avoided if non-blocking Send commands are used. This is a standard feature of MPI. However, the potential problem of a memory leak still remains. One feature of this solution is that it curtails the memory leak somewhat. Ofcourse in the ideal world, we would want this memory leak removed altogether. The reason why the number of errant Send's should be lesser is the following. In the original protocol, once a process releases the lock, it immediately issues numerous Send messages to all blocked processes. When there is high contention for locks, with multiple processes holding locks at the same time and multiple processes blocked on them, it is reasonable to assume the existance of extra Sends. However, in the "new version of the protocol, each process must bid on a blocked process before issuing a Send message. As described, the problem arises when one bidder is so slow that the previous bidder issues the Send command and replaces its bid before the other bidder even gets a chance to bid. In a system where there is high contention for locks, the probability of this happening is small. So In the worse case, the new version will have as many errant Send's as the old version. However, in the average case, the newer version will have lesser errant Sends.

6 Promela Model

Promela is a modelling language with C-like syntax. The promela model for this protocol consists of 3 files whose contents are described below.

6.1 globals.prom

This file contains all the global variables used in the model. `get_requests` and `put_requests` are two lists used to model MPI one-sided communication. The purpose of these lists is to delay MPI_Put and MPI_Get operations until the proper synchronization call. This is an overkill to the MPI standard which states that MPI one-sided operations may be completed some time before the relevant synchronization call.

`lock_data` is the public window M. The size of this array is $3 * nprocs$ to make room for flag, start and end entries for each process.

`private_data` is an $nprocs \times nprocs$ array. Even though it is a global array it is used to model the private copy of `lock_data` that each process would have.

`wait_chan` is an array of channels of size `nprocs`. A process `p` will block on a receive call from `wait_chan[p]` once it realizes it cannot enter the critical section. This is line 24 of the original protocol.

the arrays `blocked_bids` and `private_bids` are used to model the "fix".

6.2 `rma.prom`

This file implements MPI one-sided communication operations. The interesting functions defined here are `MPI_Get`, `MPI_Put`, `MPI_Get_helper` and `MPI_Put_helper`. `MPI_Get` is the standard one-sided get command. It goes through the `get_requests` list until it finds an unused request. It fills it up with all the information required to process an `MPI_Get` command. The request is then saved until `MPI_Get_helper` is called. This routine goes through the `get_requests` list and fulfills all pending `MPI_Get` requests at once. `MPI_Put` and `MPI_Put_helper` behave similarly.

6.3 `byte-range.prom`

This file contains the actual locking protocol. This looks almost exactly like the pseudo-code except for some minor simplifications. The `do-od` construct in promela is used to model the while and for loops. Each `do-od` construct contains several guarded statements of the form `guard :: statement(s)`. A guard is a boolean value. On each iteration of the `do-od` statement, all guards are evaluated and for those that evaluate to true, the corresponding statements are executed. A loop can this be simulated by having two guarded statements in a `do-od`. The first is the loop condition while the second is the loop body. The `if-fi` construct is similar to the `do-od` except it is executed only once while the `do-od` is executed repeatedly until an explicit break call.

6.4 `lock-test.prom`

This file contains the test program for the protocol. It is a fairly simple program. It contains of three processes all executing the same piece of code. Each process requests a lock on the range 3-5 once. An error is reported if any one of the processes fails to return from the `byte_range_proc_routine`.

7 Model Checking

The model works by determining a set of active statements for each process. Each active statement can be thought of as the transition for a finite automaton. The model-checkers ability to detect errors is based on the fact that it examines all possible interleavings of the statements for each process. For instance, in our "fixed" protocol, the error will not be caught if each process behaves nicely and places its bid before the bidding vector is over-written. However, in the unlikely scenario where one of the bidding processes is much slower than the others, the error is exposed. The spin model checker examines all such possibilities and reports the error.

8 A Better Solution

The main idea behind this solution is to have the blocked process pick the process it needs to wake it up before it sleeps. If we were to somehow achieve this, we would potentially eliminate our extra sends. In other words, if the blocked process picks only one process to wake it up, then only that process can send it a message and no other. However, this is contingent upon the assumption that a process can advertise its intent to the rest of the world in time.

Suppose P1 and P2 are in the critical section, P3 needs the lock that conflicts with both P1 and P2. P3 may now pick either one, say P1, to wake it up. There are two possible cases: either P1 sees this information before it releases the lock, or it doesn't see the information at all. If P1 gets a chance to see this information, everything is well and P1 sends a message to P3. P2 ignores P3 altogether. The problematic case is where P1 doesn't see this information. P2 may now potentially be blocked forever. However, our memory model allows us to write to/read from shared memory at the same time. So at the time P3 writes its intent to choose P1, it can also see P1's current state. In other words, at the time P3 writes its information, it will know whether or not P1 will be able to see it. So if P1 is going to miss this global state change, P3 may simply pick another process, P2 in this case, and repeat. In case P2 also leaves before P3 is done, P3 must loop back and try to reacquire the lock.

While this sounds good, there is still a slight possibility for an error. The assumption made in this solution is that if P3 were to pick P1 but P1 had already released the lock, that P1 would never see that information. So P3 can now safely change this information. However, it is still possible for P1 to reacquire the lock, enter the critical section, release the lock and see this information before P3 gets a chance to change it. In this case, P1 will erroneously send a message to P3. The thing to note here is that the error arises only if P1 reacquires the lock. We can fix this simply by assigning a number to every iteration of the lock-unlock cycle. If each process maintains a counter of how many times it has tried to acquire the lock, it will be trivial to distinguish different locking cycles. P3 must now pick a (process, version) pair instead of a process only. This will allow us to completely avoid extra sends. This algorithm is implemented below.

The major change in the algorithm is the size of the global array, locklist. Previously, locklist had size $3 * nprocs$, where $nprocs$ is the number of processes. Now, locklist will have size $6 * nprocs$. A section of locklist will now look like (flag, start, end, my_counter, my_pick, pick_count). Flag, start and end behave as before. my_counter is the processes' private counter. my_pick is the process it wants to wake it up while pick_count is the iteration that my_pick was in when this process made its decision.

We will also maintain a private per-process list picklist that contains all possible conflicting processes. This will help us make a decision on which process to pick.

Another change is to create new datatypes to modify locklist. The new datatype will skip the my_counter part of locklist and modify everything else.

We will call this locktype2.

```
Lock_acquire ( int start , int end)
{
    val [0] = 1; /* flag */ val [1] = start ; val [2] = end ; val [3] = -1 ; val [4]

    int ctr = 1;

    while (1) {
        /* add self to locklist */
        MPI_Win_lock ( MPI_LOCK_EXCLUSIVE , homerank , 0, lockwin );
        MPI_Put (& val , 5, MPI_INT , homerank , 6*( myrank ), 1, locktype2 , lockwin);
        /* increment counter */
        MPI_Accumulate(&ctr , 1, MPI_INT, homerank , 6 * myrank + 3,
                      1, MPI_INT, MPI_SUM, lockwin);
        MPI_Get ( locklistcopy , 6*( nprocs -1) , MPI_INT , homerank , 0, 1,
                 locktype1 , lockwin );
        MPI_Win_unlock ( homerank , lockwin );
        /* check to see if lock is already held */
        conflict = 0;
        cprocs_i = 0;

        for (i=0; i < ( nprocs - 1); i ++ ) {
            if (( flag == 1) && ( byte ranges conflict with lock request )) {
                conflict = 1;
                picklist[cprocs_i] = current_proc_pid;
                cctrs[cprocs_i] = current_proc_counter;
            }
        }
        if (conflict == 1) {

            j = 0;

            while(j < cprocs_i) {
                MPI_Win_lock ( MPI_LOCK_EXCLUSIVE , homerank , 0, lockwin );
                val [0] = 0; val [3] = picklist[j]; val [4] = cctrs[j];

                /* reset flag to 0, indicate pick and pick_counter */
                MPI_Put (&val , 5, MPI_INT , homerank , 3*( myrank ),
                        1, locktype2 , lockwin );
                MPI_Win_unlock ( homerank , lockwin );

                if(picklist[j] has left) {
                    /* repeat for the next process in picklist */
                    j++;
                }
            }
        }
    }
}
```

```

    }

    else {
        /* wait for notification from some other process */
        MPI_Recv (NULL , 0, MPI_BYTE , MPI_ANY_SOURCE , WAKEUP , comm ,
                 MPI_STATUS_IGNORE );
        /* retry the lock */
    }
}
/* if the entire list has been traversed , retry the lock */
}
else {
    /* lock is acquired */
    break ;
}
}
}

Lock_release ( int start , int end)
{
    val [0] = 0; val [1] = -1; val [2] = -1; val [3] = -1; val [4] = -1;
    /* set start and end offsets to -1, flag to 0,
       pick and pick_count to -1 and get everyone else's status */
    MPI_Win_lock ( MPI_LOCK_EXCLUSIVE , homerank , 0, lockwin );
    MPI_Put (val , 5, MPI_INT , homerank , 3*( myrank ), 1, locktype2 , lockwin );
    MPI_Get ( locklistcopy , 6*( nprocs -1) , MPI_INT , homerank , 0, 1, locktype2 );
    MPI_Win_unlock ( homerank , lockwin );

    i = 0 ; /* ranks are off by 1 because of the derived datatype */
    while ( i < ( nprocs - 1)) {
        /* the byte ranges don't even matter here , just check the pick and pick_count */
        if ( locklistcopy [6*i + 4] == myrank && locklistcopy [6*i + 5] == locklistcopy [6*i + 4])
            MPI_Send (NULL , 0, MPI_BYTE , i+1, WAKEUP , comm );
        i++;
    }
}
}

```

8.1 Efficiency

There needs to be some discussion on how efficient this algorithm is, compared to the previous algorithm. Since we are in a distributed setting, we will limit this discussion to accessing shared memory, i.e. the lock, put, get, unlock sections of code above. In the original algorithm, a process typically executes two of these sections in order to acquire the lock. In the case where it may be blocked, it executes 2 extra sections each time it is blocked. So if the cost of accessing

shared memory is C , the cost of obtaining a lock is $(2C + bC)$ where b is the number of times it is blocked.

8.1.1 Worse case analysis

Calculating total cost in the new algorithm is trickier. In case there are no conflicts, a process still needs to access shared memory only twice. In this case C changes slightly, since the shared memory region is larger and more reads/writes need to be performed. For now, we can ignore this change since the overwhelming cost is that of obtaining a lock on shared memory. What happens when there is a conflict? The process that is blocked may access shared memory b_p times, where b_p is the number of processes holding the lock that it conflicts with. In the worse case, this number may be as large as $n-1$ where n is the total number of processes. But we are not done yet. This process may not find any match even after going through the entire list. In this case, it will have to retry for the lock. This time also it may find conflicting processes and then have to retry the lock again. This is the classic problem of starvation. But, starvation is possible also in the original algorithm. So if we ignore that aspect for now, we can put an upper bound on the worse case performance of the new algorithm. This bound is $(2C + (n-1)C)$ where n is the total number of processes.

We have not yet done the worse case analysis on the original algorithm. Remembering the original bug, that there may be more than 1 Sends for a matching Receive, the worse case scenario would mean an endless pool of Sends. In this case, the blocking will not be effective at all. Each time a process tries to block on a Recv call, it will find a Send waiting for it and will simply retry the lock. Our lock has transformed into a busy-wait spin-lock instead of the intended blocked-lock. It is easy to see that the worse case for this algorithm is unbounded even without starvation.

8.1.2 Average case analysis

Now that we have performed worse case analyses on both algorithms, we can also do an average case analysis. This is always harder to do than the worse case analysis since certain assumptions must be made.

References

- [1] Rajeev Thakur, Robert Ross, Robert Latham. Implementing Byte-Range locks using MPI One-Sided Communication. *some proceedings*